A Simple XML Producer-Consumer Protocol

Warren Smith, Dan Gunter, Darcy Quesnel
wwsmith@nas.nasa.gov, dkgunter@lbl.gov, quesnel@mcs.anl.gov
Global Grid Forum Performance Working Group

1. Introduction

This document describes a simple XML-based protocol that can be used for producers of events to communicate with consumers of events. The protocol described here is not meant to be the most efficient protocol, the most logical protocol, or the best protocol in any way. This protocol was defined quickly and its intent is to give us a reasonable protocol that we can implement relatively easily and then use to gain experience in distributed event services. This experience will help us evaluate proposals for event representations, XML-based encoding of information, and communication protocols.

The next section of this document describes how we represent events in this protocol and then defines the two events that we choose to use for our initial experiments. These definitions are made by example so that they are informal and easy to understand. The following section then proceeds to define the producer-consumer protocol we have agreed upon for our initial experiments.

2. Events

There is an entire Performance Working Group document on event schemas. Since two of the authors of this document are authors of the event schema document, the events defined here closely relate to the events defined in this document. The difference is that the events defined here are meant to be very simple. It should be easy to write code to translate from programming language structures to the XML and from the XML back to programming language structures. In this document, we use the following simplifications:

- Events are not required to have explicit types that must be checked.
- Element values are always strings (follows from the previous simplification).
- Elements do not have any attributes, including accuracy or unit attributes.
- No contexts.

We assume that an event consists of a type and a set of elements where each element is a <name, value> pair and the names and values are strings. Each element can be either implicitly required or optional. We say implicitly because there is not any explicit type checking being performed at this time.

To further simplify things, we initially only handle 2 different types of events. The next two subsections describe these two types.

2.1. CPU Load Event

The CPU load event type is a simple event containing the load information returned by the Unix "uptime" command. We therefore use the event type "UptimeCPULoad" for this event to differentiate it from other means of measuring CPU load. This event must contain the following elements:

- TimeStamp. The time at which the CPU load event was generated. The format of the time contained in the value of this element is a restricted version of the ASCII format proposed by the document "A Standard Timestamp for Grid Computing " (Dan Gunter, Brian Tierney). This

standard is an extension of one of the variations of the ISO 8601 time format. To restrict the time format, we assume that all times are reported in GMT or Zulu time and that precision is not represented. An example time with milliseconds would look like: 2000-11-09T11:12:00.000Z.

- Load1. The 1-minute CPU load reported by uptime.
- Load5. The 5-minute CPU load reported by uptime.
- Load15. The 15-minute CPU load reported by uptime.
- HostName. The name of the host the load measurement is made on.

Here is an example UptimeCPULoad event in our XML encoding:

```
<UptimeCPULoad>
    <Load1>1.5</Load1>
    <Load5>1.6</Load5>
    <Load15>1.3</Load15>
    <HostName>foo.nas.nasa.gov</HostName>
    <TimeStamp>2000-11-09T21:51:45Z</TimeStamp>
</UptimeCPULoad>
```

Note that the indenting and new lines are for readability purposes only. We assume that the events will not contain any new lines or spaces around element tags when the events are transmitted.

When asking for a CPU load event, the following input parameters can be specified:

- Period. The number of seconds between each uptime measurement and event generation. This parameter is only used when a subscription is performed. If this parameter is specified for a query, it is ignored.

An example of a description of the events a consumer wants to receive is:

```
<UptimeCPULoad>
    <Period>600</Period>
</UptimeCPULoad>
```

2.2. Round Trip Time Event

The second event we define here is a latency event with data produced by the Unix ping command. We simply call this event a “Ping” event. The event must contain the following elements:

- TimeStamp. The time at which the ping was performed. The format is the same as the time stamp above.
- SourceHostName. The host name or IP address of the host that is performing the ping command.
- TargetHostName. The host name or IP address of the host that the source host is pinging. Note that this name or IP address can indicate 1 of several network interfaces on the target host.
- RoundTripTime. The round-trip time reported by the ping command in milliseconds.

Here is an example Ping event in our XML encoding:

```
<Ping>
    <SourceHostName>foo.nas.nasa.gov</SourceHostName>
    <TargetHostName>bar.lbl.gov</TargetHostName>
    <RoundTripTime>7</RoundTripTime>
    <TimeStamp>2000-11-09T21:53:45Z</TimeStamp>
</Ping>
```

When asking for a ping event, the following input parameters can be specified:

- Period. The number of seconds between each uptime measurement and event generation. This parameter is only used when a subscription is performed. If this parameter is specified for a query, it is ignored.
- TargetHostName. The name or IP address of the host that will be pinged.

An example of a description of which events a consumer desires is:

```
<UptimeCPULoad>
    <Period>600</Period>
    <TargetHostName>bar.lbl.gov</TargetHostName>
</UptimeCPULoad>
```

3. XML Protocol

This section describes the XML protocol we use for communication between producers and consumers. We require that our protocol allow:

- Consumers to subscribe for events
- Consumers to unsubscribe from events
- Producers to send events to consumers
- Consumers to query for a single event

We assume that:

- TCP sockets are used for communication.
- No authentication, authorization, or security is supported.
- XML is used to represent control and data messages.
- One socket between a producer and a consumer is used for both control and data messages.
- Multiple subscriptions can be active over a socket.
- Multiple requests can be outstanding on the same connection.

Given the requirements of our protocol, we define the following messages:

1. SubscribeRequest
2. SubscribeReply
3. UnsubscribeRequest
4. UnsubscribeReply
5. Event
6. QueryRequest
7. QueryReply

Note that many of the messages are pairs of request/reply messages.

3.1. General Message Format

In general, each message consists of:

1. Length: the number of bytes in the message as a 32-bit integer in network byte order (C functions: htonl, ntohl)
2. The XML tags that indicate the message type. For example: <SubscribeRequest> … </SubscribeRequest> or <QueryReply> … </QueryReply>
   2.1. Request messages always have a requester-unique request ID chosen by the requester. This request ID is an attribute of the message tag. For example: <SubscribeRequest requestID=”1”> … </SubscribeRequest>.
   2.2. Reply messages always have a request ID. The request ID for a reply matches the request ID given by the requester for the request that is being replied to. For example, a reply to the above subscribe request would be: <SubscribeReply requestID=”1”> … </SubscribeReply>.
   2.3. Reply messages always have a return code and may have a return message. The return code indicates if an operation was successful (0) or a failure (1). These return codes may be expanded later to contain different non-zero error codes. The return message is a text message that contains detailed user-readable information about the status of a request.
3. The message-specific data inside the XML tags that identify the message. For example: <SubscribeRequest requestID=”1”> <UptimeCPULoad> … </UptimeCPULoad> </SubscribeRequest>

3.2. Subscribing for Events

When a consumer wants to receive a stream of events from a producer, it subscribes for events. For now we are assuming that after a subscription, a producer pushes events to the consumer that has subscribed for them. This subscription takes the form of a SubscribeRequest message from the consumer to the producer, followed by a SubscribeReply message from the producer to the consumer.

3.2.1. SubscribeRequest Message

The subscribe request message consists of:

- Event type (required).
- Any input parameters needed to generate events (optional).
- Request ID so that a reply can be matched with a request.

Here are two examples of SubscribeRequest messages:

```
<SubscribeRequest requestID=”1”>
    <CPULoadParameters>
        <Period>600</Period>
    </CPULoadParameters>
</SubscribeRequest>


<SubscribeRequest requestID=”2”>
    <PingParameters>
        <Period>300</Period>
        <TargetHostName>bar.lbl.gov</TargetHostName>
    </PingParameters>
</SubscribeRequest>
```

Note: In the future, we will probably want an event filter. After a producer generates an event, a filter is used to determine if the event should be sent to a consumer. For example, a filter may indicate that an

event should be sent only if the load is greater than 5.0. The filter is a Boolean expression of event element names and constants that is applied to an event. We think the filter specification should go inside the PingParameters.

3.2.2. SubscribeReply Message

The subscribe reply message consists of:

- ReturnCode (required). 0 means success, 1 means failure. More non-zero return codes to represent more detailed failures may be added in the future.
- ReturnMessage (optional). Text giving further information about the successful or unsuccessful subscribe.
- Request ID of the request that this message is in reply to.
- An optional producer-unique SubscriptionID that identifies the subscription that was successfully made by the consumer (if one was). The subscription ID should be present if the subscription was successful and should not be present if the subscription was not successful. The subscription ID is used so that a consumer can easily route an event to the handler for events from that subscription.

Two examples of SubscribeReply messages are:

```
<SubscribeReply requestID=”1”>
    <ReturnCode>1</ReturnCode>
    <ReturnMessage>The period specified is too small.</ReturnMessage>
</SubscribeReply>


<SubscribeReply requestID=”2”>
    <ReturnCode>0</ReturnCode>
    <SubscriptionID>1234</SubscriptionID>
</SubscribeReply>
```

3.3. Unsubscribing from Events

When a consumer wishes to stop receiving a stream of events from a producer, it unsubscribes from those events. This unsubscription is accomplished by the consumer sending an UnsubscribeRequest message to the producer and the producer sending an UnsubscribeReply message back to the consumer in response to the request.

3.3.1. UnsubscribeRequest Message

The unsubscribe request message consists of:

- The RequestID of the request that this message is in reply to.
- The SubscriptionID that identifies the subscription to be unsubscribed from.

An example of an UnsubscribeRequest message is:

```
<UnsubscribeRequest requestID=”9”>
    <SubscriptionID>1234</SubscriptionID>
</UnsubscribeRequest>
```

3.3.2. UnsubscribeReply Message

The unsubscribe reply message consists of:

- ReturnCode (required). 0 means success, 1 means failure. More non-zero return codes to represent more detailed failures may be added in the future.
- ReturnMessage (optional). Text giving further information about the successful or unsuccessful subscribe.
- Request ID of the request that this message is in reply to.

Example of an UnsubscribeReply message are:

```
<UnsubscribeReply requestID=”9”>
    <ReturnCode>0</ReturnCode>
</UnsubscribeReply>


<UnsubscribeReply requestID=”9”>
    <ReturnCode>1</ReturnCode>
    <ReturnMessage>Unknown subscription ID.</ReturnMessage>
</UnsubscribeReply>
```

3.4. Event Message

An event message is sent from the producer to the consumer after a consumer has subscribed for events. An event message consists of:

- A subscription ID to identify which subscription the event is for
- The event data in the format described in Section 2.

Example Event messages are:

```
<Event subscriptionID=”1234”>
    <UptimeCPULoad>
        <Load1>1.5</Load1>
        <Load5>1.6</Load5>
        <Load15>1.3</Load15>
        <TimeStamp>2000-11-09T21:51:45Z</TimeStamp>
    </UptimeCPULoad>
</Event>


<Event subscriptionID=”1235”>
    <Ping>
        <SourceHostName>foo.nas.nasa.gov</SourceHostName>
        <TargetHostName>bar.lbl.gov</TargetHostName>
        <RoundTripTime>7</RoundTripTime>
        <TimeStamp>2000-11-09T21:53:45Z </TimeStamp>
    </Ping>
</Event>
```

3.5. Querying for an Event

We believe there will be many cases when a consumer wants to ask for just 1 event from a producer. Instead of having a consumer subscribe, receive 1 event, and then unsubscribe, we allow a consumer to

query a producer for an event. This query consists of a QueryRequest message that a consumer sends to the producer and a QueryReply message that the producer sends to the consumer in response to the QueryRequest message.

3.5.1. QueryRequest Message

The query request message is very similar to the subscribe request message and consists of:

- Event type (required).
- Any input parameters needed to generate events (optional).
- Request ID so that a reply can be matched with a request.

Here are two examples of QueryRequest messages:

```
<QueryRequest requestID=”15”>
    <UptimeCPULoadParameters>
    </UptimeCPULoadParameters>
</QueryRequest>


<QueryRequest requestID=”20”>
    <PingParameters>
        <TargetHostName>bar.lbl.gov</TargetHostName>
    </PingParameters>
</QueryRequest>
```

3.5.2. QueryReply Message

The QueryReply messages are similar to the event messages and consist of:

- A request ID to identify which QueryRequest this reply is for
- The event data in the format described in Section 2.

Example QueryReply messages are:

```
<QueryReply requestID=”15”>
    <UptimeCPULoad>
        <Load1>1.5</Load1>
        <Load5>1.6</Load5>
        <Load15>1.3</Load15>
        <TimeStamp>2000-11-09T21:51:45Z</TimeStamp>
    </UptimeCPULoad>
</QueryReply>


<QueryReply requestID=”20”>
    <Ping>
        <SourceHostName>foo.nas.nasa.gov</SourceHostName>
        <TargetHostName>bar.lbl.gov</TargetHostName>
        <RoundTripTime>7</RoundTripTime>
        <TimeStamp>2000-11-09T21:53:45Z </TimeStamp>
    </Ping>
</QueryReply>
```

4. Summary

This document describes a simple XML publish-subscribe protocol for performance events. The protocol described here is not meant to be the most efficient protocol, the most logical protocol, or the best protocol in any way. However, it provides a foundation of practical experience for evaluating alternate protocol implementations, including higher-level solutions such as XML-RPC (http://www.xml-rpc.com) and SOAP (http://www.develop.com/soap).

If you have comments on this paper, you can contact the authors or the Grid Forum Performance Working Group at perf-wg@gridforum.org. The Grid Forum is an open community; more information on current activities and how to join is available at the main web site (http://www.gridforum.org) and the Performance Working Group site (http://www-didc.lbl.gov/GridPerf/).

5. Acknowledgements

Input from many people went into this document, including many of the attendees of the Grid Forum meetings. The LBNL portion of this work is supported by the U.S. Department of Energy, Office of Science, Office of Computational and Technology Research, Mathematical, Information, and Computational Sciences Division, under contract DE-AC03-76SF00098 with the University of California.